ARMY RESEARCH LABORATORY

Compilation of a Network Security/Machine Learning Toolchain for Android ARM Platforms

by Ralph P. Ritchey, Garrett S. Payer, and Dr. Richard E. Harang

ARL-CR-0739 **July 2014**

Prepared by

ICF International
7125 Thomas Edison Dr Ste 100
Columbia, MD 21046

Under contract

W911QX-12-F-0052

Approved for public release; distribution unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-CR-0739 **July 2014**

Compilation of a Network Security/Machine Learning Toolchain for Android ARM Platforms

Ralph P. Ritchey, Garrett S. Payer, and Dr. Richard E. Harang
Computational and Information Sciences Directorate, ARL

Prepared by

ICF International
7125 Thomas Edison Dr Ste 100
Columbia, MD 21046

Under contract

W911QX-12-F-0052

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
July 2014		

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Compilation of a Network Security/Machine Learning Toolchain for Android ARM Platforms	W911WX-12-F-0052
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
Ralph P. Ritchey, Garrett S. Payer, and Dr. Richard E. Harang	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
U.S. Army Research Laboratory ATTN: RDRL-CIN-D 2800 Powder Mill Road Adelphi, MD 20783-1197	ARL-CR-0739
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report provides the instructions necessary to prepare the Basic Linear Algebra Subprograms (BLAS) library and LibPCap library for use on an Android-based device. Instructions are also included that give the additional capability of being able to compile FORTRAN-based source code to the Android Native Development Kit (NDK) that is not provided by default, which will be needed to compile BLAS. These packages provide basic functionality for machine learning-oriented network security applications, and promote the use of the Android platform as a suitable test bed for research into security tools for mobile and ad-hoc networks.

15. SUBJECT TERMS

Android Toolchain Fortran Security Machine Learning Libpcap pcap

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
A. Report	b. ABSTRACT	c. THIS PAGE			Ralph P. Ritchey
Unclassified	Unclassified	Unclassified	UU	24	19b. TELEPHONE NUMBER (*Include area code*) (301) 394-0780

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

1. Background **1**

2. Configuration Used **2**

3. Android NDK FORTRAN Compiler Support **2**

4. Compiling the BLAS Library for Android SDK (Static Library) **5**

5. Compiling the LibPCap Library for Android SDK (Static Library) **7**

6. Example Application: Compilation **8**

7. Example Application: Installation and Execution on an Emulated Android Device **8**

8. References **10**

Appendix A. fortran4android **11**

Appendix B. test_application **15**

Distribution List **17**

INTENTIONALLY LEFT BLANK.

1. Background

With the introduction and acceptance of mobile technology into every aspect of daily life, there is a strong desire to push towards the official use of these devices and the capabilities they provide in the tactical environment. Just as conventional computer systems need to be protected, mechanisms need to be developed to protect these mobile devices. Due to the inherently mobile nature of these devices, and corresponding lack of fixed infrastructure that can be used to effectively protect them, the first line of defense for such devices must be on the device, itself. As many of these devices are designed to be lightweight, small, and have a low power draw, their ability to execute complex and resource intensive algorithms is limited. These resource constraints have led to an interest in lightweight machine learning techniques for providing such defenses (see, e.g., [*1*]). In contrast to most conventional approaches, such as signature-based methods, these techniques typically can be constructed to allow for a flexible tradeoff between speed, accuracy, and memory, allowing the algorithm to be fine-tuned to the resources and criticality of device or platform being protected. Furthermore, in many cases, machine learning techniques generate complex and non-human-readable internal states (*2*), which may offer operational security benefits that signature-based systems often cannot.

In order to provide machine learning-based protection services for mobile devices deployed into a tactical environment using the Android operating system (currently the most popular mobile OS), several key libraries are needed. Basic Linear Algebra Subprograms (BLAS) (*3*), which provides a fast linear algebra library, is needed for machine learning-based algorithms, which rely heavily on inner product operations, as do most linear and kernel-based classifiers (including ELIDe, referenced previously). LibPCap (*4*), the de facto standard library used for performing network packet capture, and is needed for enabling a device to monitor live network traffic or process previously captured network traffic that was saved to a file. The incorporation of such standard libraries into the Android framework will enable rapid transition of novel algorithms to Android devices.

We first describe the configuration of the build environment and then discuss the process of modifying FORTRAN to interoperate with the Android Native Development Kit (NDK) such that BLAS can be compiled. We then provide instructions for compiling LibPCap onto the Android architecture. In the final section of this report, the code and compilation process used to build a small application to test the libraries are provided. The instructions and techniques used for this application could be used as a basis to build other, more sophisticated command line type applications for use in an Android environment.

2. Configuration Used

The following is a list of the software and hardware used while getting the libraries compiled and FORTRAN support added. This system configuration is much larger than what is actually needed to complete the tasks outlined in this report, so use of a smaller system should suffice.

- Operating System: Redhat Enterprise Linux (RHEL) version 6.5
- Android Development Tools (ADT): version 22.3.0-887826
- Dell Optiplex 960
  - 8GB Memory
  - Intel Core2 CPU
    - Quad Core
    - 3.00GHz

3. Android NDK FORTRAN Compiler Support

The Android NDK (*5*) allows developers to include "native" code (C/C++) in their Android projects so it can be accessed via the Java Native Interface (JNI) (*6*) from their Android application code. This capability can be very useful when a needed library is available, but porting it to Java will require a considerable amount of time and effort.

While the default capabilities of the NDK are useful, there are situations where the ability to compile FORTRAN-based code is desirable. In order to support this effort, extra steps must be taken in order to modify the default NDK to support the FORTRAN programming language. The basis used for the steps provided in this technical report was found on "Danilo's Tech Blog" (*7*). The blog entry provides updated patches and a shell script that automates the list of manual steps provided at the "Specific Impulses" (*8*) blog.

It should be noted that the patches provided at "Danilo's Tech Blog" are for version "r9" of the Android NDK. At the time of writing, "r9c" is available; however, the patches and compilation process do not complete cleanly. Therefore, it is necessary to specifically use the "r9" version of NDK and the versions specified for other required components until newer patches are available.

The first goal in adding FORTRAN support to the NDK is to obtain and install the correct version:

1. Download the "r9" version of the NDK: http://dl.google.com/android/ndk/android-ndk-r9-linux-x86_64.tar.bz2

2. Decompress and untar the downloaded file. Note where it is located—when you call the `ndk-build` script, you will need to specify the full path. The script uses the path it was called from to dynamically configure its environment when run. (It may be possible to add the script's path to `$PATH`; be aware, however, that there may be a slight chance it will not work correctly, as it dynamically configures its runtime environment based on how it was initially executed. If a problem is encountered, try re-running it and specifying the full path.)

3. Download both the shell script and patch file from the "Danilo's Tech Blog" Web site, placing both files into the base directory of the NDK (`android-ndk-r9`).

Now that the NDK has been installed, the next goal is to add FORTRAN support:

1. The first step is the installation of a script (`fortran4android`) that automates the patching needed for building the FORTRAN compiler and the compilation process of the FORTRAN compiler. There are two options available:

   a. *Option 1:* Create a copy of the script provided in appendix A, which is an already modified version of the script provided at "Danilo's Tech Blog" (*7*), in the base directory of the NDK (`android-ndk-r9`). Download the patch file and install it into the same directory as the script.

   b. *Option 2:* Download the shell script and patch file from the "Danilo's Tech Blog" Web site, placing both files in the base directory of the NDK (`android-ndk-r9`).

      i. Edit the shell script and either comment out or remove all bits of code that install (`yum install`) packages or retrieve them from outside sources '`wget`, `svn`, etc.). You will manually obtain them later in another step from appropriate sources.

      ii. Near the end of the script where the compilation is performed are two `if` statements that will either skip building the toolchain ("`... toolchain appears to be already present.. skipping`") or copying the toolchain config file ("`... toolchain config files already present.. skipping`"). Comment out the '`if/then`' portion *but leave the contents of the '`else`' intact so they will be performed*.

2. Within the NDK base directory, create a `src` subdirectory.

3. Under the `src` directory, create the following subdirectories and obtain/decompress/untar the software with the *specific version number*, as listed for each. Most can be obtained from the official GNU site; however, a few will need to be obtained from an official Fedora Web site.

Directory	Version	Obtain From
binutils	binutils-2.22.90	ftp://ftp.gnu.org
build		git clone https://android.googlesource.com/toolchain/build build
cloog	cloog-0.19.0	pkgs.fedoraproject.org
expat	expat-2.0.1	sourceforge.net
gcc	gcc-4.8.0	ftp://ftp.gnu.org
gdb	gdb-7.4.1	ftp://ftp.gnu.org
gmp	gmp-5.0.5	ftp://ftp.gnu.org
isl	isl-0.11.1	pkgs.fedoraproject.org
mpc	mpc-1.0.2	ftp://ftp.gnu.org
mpfr	mpfr-3.0.1	ftp://ftp.gnu.org
ppl	ppl-0.11.2	pkgs.fedoraproject.org

.

4. From the base of the NDK directory, execute the `fortran4android` script. Note the following items:

   a. If the script errors in regard to a few subdirectories not existing, create them manually and then re-execute the script. Examples may include `toolchains/arm-linux-androideabi-4.8.0` and `toolchains/arm-linux-androideabi-4.8.0/prebuilt`.

   b. When you apply the patches contained in the `ndk-r9-fortran.patch` file, the patch application process may report the application of several patch hunks failing. It will notify you where the rejected hunks were written out, allowing you to manually verify if they were applied or not. In most cases they were, although there may be one or two that did not. (When manually verifying the application of the patches, you will not be able to go by the line numbers contained in the rejected patch file. You will need to search for the location where the patch should have been applied based on code surrounding the actual patch.)

4. Compiling the BLAS Library for Android SDK (Static Library)

Now that the NDK has been downloaded, installed, and updated to support compiling FORTRAN code, it will now be possible to compile the FORTRAN-based BLAS library. The following steps provided will result in a statically compiled library. The first goal is setting up the proper directory structure and obtaining the BLAS library:

1. The compilation of the BLAS library needs to take place in a specifically named directory. This directory can be included as part of an existing Android application project or it can be done separately:
   a. *As part of existing Android project.* Within the projects base directory, create a subdirectory named '`jni`'.
   b. *Not as part of Android project.* Create a `jni` directory in your preferred location.
2. Download the BLAS library (version marked "LAST UPDATE: Tuesday Apr 19th 2011") into a directory of your choice and decompress it, which will result in a subdirectory named `BLAS`.
3. From the `BLAS` directory, copy all of the `*.f` files to the `jni` subdirectory created in Step 1.
4. Within the `jni` directory, create an `Android.mk` file with the following contents:

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)

# LOCAL_ALLOW_UNDEFINED_SYMBOLS := true
LOCAL_LDLIBS += -lgfortran
LOCAL_MODULE := blas_LINUX
LOCAL_SRC_FILES := caxpy.f chemm.f cscal.f ctpsv.f dgbmv.f dscal.f dsyr2k.f
dzasum.f scasum.f srot.f ssymm.f strmm.f zdscal.f zher2.f zsyr2k.f ccopy.f
chemv.f csrot.f ctrmm.f dgemm.f dsdot.f dsyrk.f dznrm2.f scnrm2.f srotg.f
ssymv.f strmv.f zgbmv.f zher2k.f zsyrk.f cdotc.f cher.f csscal.f ctrmv.f
dgemv.f dspmv.f dtbmv.f icamax.f scopy.f srotm.f ssyr.f strsm.f zgemm.f
zherk.f ztbmv.f cdotu.f cher2.f cswap.f ctrsm.f dger.f dspr.f dtbsv.f
idamax.f sdot.f srotmg.f ssyr2.f strsv.f zgemv.f zhpmv.f ztbsv.f cgbmv.f
cher2k.f csymm.f ctrsv.f dnrm2.f dspr2.f dtpmv.f isamax.f sdsdot.f ssbmv.f
ssyr2k.f xerbla.f zgerc.f zhpr.f ztpmv.f cgemm.f cherk.f csyr2k.f dasum.f
drot.f dswap.f dtpsv.f izamax.f sgbmv.f sscal.f ssyrk.f zaxpy.f zgeru.f
zhpr2.f ztpsv.f cgemv.f chpmv.f csyrk.f daxpy.f drotg.f dsymm.f dtrmm.f
lsame.f sgemm.f sspmv.f stbmv.f zcopy.f zhbmv.f zrotg.f ztrmm.f cgerc.f
chpr.f ctbmv.f dcabs1.f drotm.f dsymv.f dtrmv.f sasum.f sgemv.f sspr.f
stbsv.f zdotc.f zhemm.f zscal.f ztrmv.f cgeru.f chpr2.f ctbsv.f dcopy.f
drotmg.f dsyr.f dtrsm.f saxpy.f sger.f sspr2.f stpmv.f zdotu.f zhemv.f
zswap.f ztrsm.f chbmv.f crotg.f ctpmv.f ddot.f dsbmv.f dsyr2.f dtrsv.f
scabs1.f snrm2.f sswap.f stpsv.f zdrot.f zher.f zsymm.f ztrsv.f

include $(BUILD_STATIC_LIBRARY)
```

Now that the BLAS library has been downloaded and the initial setup completed, it can now be compiled:

1. Change your working directory into the `jni` subdirectory.
2. Execute the Android NDK build process by executing the following command: `<path_to_Android_NDK)/ndk-build`.
3. A listing of the files being compiled will scroll by. The final line of output should be: "`StaticLibrary  : libblas_LINUX.a`".
4. The static library file will be contained under: `<path_to_jni_directory>/obj/local/armeabi/`.

Now that the compilation of the library has been completed, the library can be copied and included in other projects as needed. Refer to the section "Example Application: Compilation" for a very simple example program that was used to test the library.

With one minor change to the Android.mk file (`BUILD_STATIC_LIBRARY` to `BUILD_SHARED_LIBRARY`), it may be possible to build a dynamically compiled library; however, that result has not been tested.

5. Compiling the LibPCap Library for Android SDK (Static Library)

The following steps describe the process needed to compile LibPCap for use as a statically compiled library on an Android device.

The first goal will be to obtain the source code for the library and set up the compilation environment:

1. The compilation of the LibPCap library needs to take place in a specifically named directory. This directory can be included as part of an existing Android application project or it can be done separately:
    a. *As part of existing Android project:* Within the projects base directory, create a subdirectory named `jni`.
    b. *Not as part of Android project:* Create a `jni` directory in your preferred location.

2. To be able to compile and utilize the LibPCap network traffic capture library, check out a version of the source code from Google's source code repository:

```
git clone https://android.googlesource.com/platform/external/libpcap
   <path_to>/jni
```

3. By default, the `Android.mk` file include with the git repository builds a static library. If a shared library is desired, edit the last line in the file and change `BUILD_STATIC_LIBRARY` to `BUILD_SHARED_LIBRARY`.

4. From within the `jni` directory execute: `<path_to_NDK>/ndk_build`.

5. Once the compilation process has been started, the list of files will scroll by as they are compiled. Depending on what library type (static, shared) is being compiled, the final library will be located in a different location:
    a. *Static library:*
    `<directory_containing_jni_directory>/obj/local/armeabi/`
    b. *Shared library:*
    `<directory_containing_jni_directory>/libs/armeabi/`

Once the compilation process has been completed, you may then copy the resulting library file to wherever it is needed. If a shared library is built and the compilation directory (`jni`) is located within the Android application project directory, the shared library file will automatically be included as part of the Android app when it is deployed to an emulator or device.

6. Example Application: Compilation

In this section, the small test application is used to verify that both the BLAS and LibPCap static libraries that resulted from the previous sections function properly. The steps and code could be used to form the basis of a more sophisticated command line application if desired.

1. Create a `jni` directory, which will contain the test application's code, the libraries compiled in previous sections, and the makefile.
2. In the `jni` directory, create the `test_application.c` file and `Android.mk` file, as shown in Appendix B.
3. Copy the `libblas_LINUX.a` file created in the 'Compiling the BLAS Library for Android SDK (Static Library)' section into the `jni` directory.
4. Copy the `libpcap.a` file created in the 'Compiling the LibPCap Library for Android SDK (Static Library)' section, as well as the `pcap.h` and `pcap-bpf.h` files provided when the source code for LibPCap was downloaded into the `jni` directory.
5. If not already in the `jni` directory, change your working directory into it.
6. From within the `jni` directory, execute: `<path_to_NDK>/ndk_build`.
7. After the compilation has completed, the executable binary `test_application` will be automatically copied to the `libs/armeabi` directory. (`libs` is contained in the same directory containing your `jni` directory and will be created automatically if it doesn't already exist.)

7. Example Application: Installation and Execution on an Emulated Android Device

In this final section, the test application created in the previous section will be copied to an emulated Android device and executed. The technique used to copy and execute the application in an emulated device is not specific to this application and may be useful for other applications.

For this section, it is assumed that the Android Developer Tools (ADT) (*9*) is installed and is functioning correctly, and that a virtual device in the supplied emulator has been created and functions. If additional documentation regarding ADT and the emulator is needed, or a deeper understanding of the techniques and commands used here is desired, the book *Android Developer Tools Essentials* (*10*) contains documentation on the use of ADT, developing apps

and using the emulator that can be highly useful. Information on their installation, configuration, and use is also readily available on the Web.

1. Using the Android Debug Bridge (ADB) supplied with ADT, open a shell on the emulated device:
   - `adb shell`
2. Remount the filesystem so the example application can be copied onto it:
   - `mount –rw –o remount rootfs /`
3. Exit the ADB shell:
   - `exit`
4. Copy the test application onto the emulated device:
   - `adb push test_application /`
5. Open a shell again on the emulated device:
   - `adb shell`
6. Run the test application:
   - `./test_application`

After the test application has completed running, the following information will be displayed:

```
root@generic:/ # ./test_application
 BLAS Test:  The dot product is: 32.000000
 pcap test:  Device: eth0
```

8. References

1. Chang, R. J.; Harang, R. E.; Payer, G. S. *Extremely Lightweight Intrusion Detection (ELIDe)*; arl-cr-0730; U.S.Army Research Laboratory: Adelphi MD, 2013.
2. Harang, R. *Bridging the Semantic Gap: Human Factors in Anomaly-Based Intrusion Detection Systems*. in Network Science and Cybersecurity, New York, Springer , 2014, pp. 15–37.
3. BLAS (Basic Linear Algebra Subprograms), [Online]. Available: http://www.netlib.org/blas/. [Accessed 13 March 2014].
4. TCPDump & LibPCAP, [Online]. Available: http://www.tcpdump.org/. [Accessed 13 March 2014].
5. Android NDK, [Online]. Available: https://developer.android.com/tools/sdk/ndk/index.html. [Accessed 11 March 2014].
6. Java Native Interface, [Online]. Available: http://docs.oracle.com/javase/6/docs/technotes/guides/jni/. [Accessed 11 March 2014].
7. Giulianelli, D. How to Build the gcc Fortran Cross-Compiler for Android (ARM and X86), 12 February 2013. [Online]. Available: http://danilogiulianelli.blogspot.com/2013/02/how-to-build-gcc-fortran-cross-compiler.html. [Accessed 12 March 2014].
8. Long, M. Android Fortram Step-by-Step Part 2: Building a Custom GCC Toolchain, 11 September 2012. [Online]. Available: http://specificimpulses.blogspot.com/2011/10/android-fortran-step-by-step-part-2.html. [Accessed 12 March 2014].
9. Developer Tools, [Online]. Available: http://developer.android.com/tools/index.html. [Accessed 14 March 2014].
10. Wolfson, M. Android Developer Tools Essentials, O'Reilly Media, Inc., 2013.
11. BLAS/LAPACK c++ tutorial/reference?, [Online]. Available: http://ubuntuforums.org/showthread.php?t=1740797&s=f9c263cef5e57d31af4c70c47e17595f. [Accessed 14 March 2014].

Appendix A. fortran4android

The following shell script is adapted from the script provided at "Danilo's Tech Blog" (*7*). The script was modified to remove the automated installation of several operating system packages and the downloading of additional dependencies not provided or made available by the operating system. This was done to ensure the additional dependencies were downloaded from more secure, reputable sites.

```
#!/bin/ksh
PROGNAME=${0##*/}
TRUE=1
FALSE=0
DEBUG="${FALSE}"
VERBOSE="${FALSE}"
export TMPDIR="${TMPDIR:-/tmp}"
TMPFILE="$TMPDIR/tmp${$}.tmp"
VERSION=1.0

function usage {
    print ""
    [[ "$1" != "" ]] && print "You forgot to pass $1 parameter to ${PROGNAME}."
    print ""
    print "Usage: ${PROGNAME} [-dvV]"
    print ""
    print "    Where -d =  debug mode"
    print "          -v =  verbose mode"
    print "          -V =  print version number and exit"
    print ""
}

function clean_up {
    rm -rf ${TMPFILE}
}

while getopts ":dvV" OPTION
do
    case "${OPTION}" in
        'd') DEBUG="${TRUE}" ;;
        'v') VERBOSE="${TRUE}" ;;
        'V') print -u2 "${PROGNAME}: version ${VERSION}" && exit 1 ;;
        '?') usage && exit 1 ;;
    esac
done

shift $(( ${OPTIND} - 1 ))

trap "clean_up" EXIT

(( VERBOSE == TRUE )) && set -x
BINUTILS_VERSION=2.22.90
GMP_VERSION=5.0.5
MPFR_VERSION=3.0.1
MPC_VERSION=1.0.2
GDB_VERSION=7.4.1
EXPAT_VERSION=2.0.1
ANDROID_NDK="android-ndk-r9"
```

```
ANDROID_NDK_VERSION="r9"
ANDROID_NDK_ROOT=/home/pritchey/bin/$ANDROID_NDK
PATCH_REPOS="./"
# set how many jobs to use to build the toolchain
NJOBS=1
# turn off expat for now...
BUILD_EXPAT="false"
BASE_PATH=$PWD

# check our OS.. only tested on Ubuntu 12.04 and Fedora 16/17
GT=`grep -c Ubuntu /etc/issue`

if [ $GT -ge 1 ]
then
    echo "... detected Ubuntu"
    echo "... only tested on 12.04.. YMMV"
    OS_TYPE="Ubuntu"
elif [ -f /etc/redhat-release ]
then
    echo "... detected Redhat derived OS"
    echo "... only tested on Fedora 16 and 17.. YMMV"
    OS_TYPE="Redhat"
else
    echo "... unsupported OS type!"
    usage && exit 1
fi

# check for 64 bit install
OS_ARCH=`uname -p`

# test to make sure we're in the right place..
if [[ "$PWD" =~ "$ANDROID_NDK" ]]
then
    echo "OK.. looks like we're in the right place"
else
    if [ -d ./$ANDROID_NDK ]
    then
        echo "... found NDK in current directory.. continuing"
        cd $ANDROID_NDK
    elif [ -f android-ndk-${ANDROID_NDK_VERSION}-linux-x86.tar.bz2 ]
    then
        echo "... extracting existing NDK archive"
        tar -jxvf android-ndk-${ANDROID_NDK_VERSION}-linux-x86.tar.bz2 > /dev/null
        cd $ANDROID_NDK
    else
        echo "...extracting downloaded NDK archive"
        tar -jxvf android-ndk-${ANDROID_NDK_VERSION}-linux-x86.tar.bz2 > /dev/null
        cd $ANDROID_NDK
    fi
fi

if [[ "$PWD" =~ "/$ANDROID_NDK" ]]
then
    echo "...NDK acquired.. continuing"
    ANDROID_NDK_ROOT=$PWD
else
    echo "...can't get into the NDK install directory.. stopping"
    exit 1
fi

SOURCE_PATH=$ANDROID_NDK_ROOT/src

if [ -d $SOURCE_PATH ]
```

```
then
    echo "src directory exists.."
else
    echo "making src directory.."
    mkdir $SOURCE_PATH
fi

cd $SOURCE_PATH

# apply fortran patch
set -x
cd $ANDROID_NDK_ROOT
patch -bNp0 < $PATCH_REPOS/ndk-${ANDROID_NDK_VERSION}-fortran.patch

# run the build..

for toolchain in arm-linux-androideabi-4.8.0 x86-4.8.0
do
    short_toolchain=$(echo $toolchain | sed 's/-4.8.0//')
    cd $ANDROID_NDK_ROOT

    if [ $OS_ARCH == "x86_64" ]
    then
        ./build/tools/build-gcc.sh --try-64 $PWD/src $PWD -j$NJOBS $toolchain
    else
        ./build/tools/build-gcc.sh $PWD/src $PWD -j$NJOBS $toolchain
    fi

    echo "... copying toolchain config files from 4.6 compiler"
    cp toolchains/${short_toolchain}-4.6/config.mk toolchains/$toolchain/.
    cp toolchains/${short_toolchain}-4.6/setup.mk toolchains/$toolchain/.
    cd toolchains/$toolchain/prebuilt
    if [ -d linux-x86_64 ]
    then
        echo "... symlinking 4.8.0 toolchain linux-x86_64 to linux-x86"
        if [ -L linux-x86 ]
        then
            echo "... symlink already exists.. skipping"
        else
            ln -s linux-x86_64 linux-x86
        fi
    fi
done
echo "Done."
exit 0
```

INTENTIONALLY LEFT BLANK.

Appendix B. test_application

The following C source code was used to test the BLAS and LibPCap libraries to ensure they worked on an Android device. The section of code used for testing the BLAS library was found on the Ubuntu Forums Web site (*11*), posted by 3Miro.

Android.mk:

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := blas_LINUX
LOCAL_SRC_FILES := libblas_LINUX.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := pcap
LOCAL_SRC_FILES := libpcap.a
include $(PREBUILT_STATIC_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE    := test_application
LOCAL_SRC_FILES := test_application.c
LOCAL_STATIC_LIBRARIES := blas_LINUX pcap
include $(BUILD_EXECUTABLE)
```

test_application.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "pcap.h"

double ddot_( const int *N, const double *a, const int *inca, const
double *b, const int *incb );

int main( int argc, char** argv ){
  double *a = (double*) malloc( 3 * sizeof(double) );
  a[0] = 1.0; a[1] = 2.0; a[2] = 3.0;
  double b[3] = { 4.0, 5.0, 6.0 };

  int N = 3, one = 1; // one really doesn't look good in C

  double dot_product = ddot_( &N, a, &one, b, &one );
  printf("\n BLAS Test:  The dot product is: %f \n",dot_product );

  char *dev, errbuf[PCAP_ERRBUF_SIZE];

  dev = pcap_lookupdev(errbuf);
  if (dev == NULL) {
    fprintf(stderr, "\n pcap test:  Couldn't find default device:
%s\n", errbuf);
    return(2);
  }
```

```
printf("\n pcap test:  Device: %s\n", dev);
return 0;
```

1 (PDF) DEFENSE TECH INFO CTR
ATTN DTIC OCA

2 (PDF) US ARMY RSRCH LABORATORY
ATTN IMAL HRA MAIL & RECORDS MGMT
ATTN RDRL CIO LL TECHL LIB

1 (PDF) GOVT PRNTG OFC
ATTN A MALHOTRA

1 (PDF) US ARMY CYBER COMMAND
ATTN 24 C PRESSLEY

8 (PDF) US ARMY RDECOM CERDEC
ATTN RDER IWI
G BERTOLI
K BOYLE
ATTN RDER IWI SP
P ROBB
S BLAIR
ATTN RDER STI IS
J SANTOS
ATTN RDER STI
S LUCAS
ATTN RDER STI TN
G ZIGLICH
M MAGENHEIMER

5 (PDF) US ARMY RSRCH LAB
ATTN RDRL CIN D
B RESCHLY
J COLE
T PARKER
ATTN RDRL CIN S
C SMITH
ATTN RDRL CIN D
L M MARVEL

2 (PDF) US ARMY RSRCH LAB
ATTN RDRL SLE I
A REVILLA
D LANDIN

10 (PDF) US ARMY RSRCH LAB
ATTN RDRL CIN
A KOTT
ATTN RDRL CIN
D KELLY
ATTN RDRL CIN D
W GLODEK
R ERBACHER
H CAM
S HUTCHINSON
R HARANG
R RITCHIE
P GUARINO
ATTN RDRL CIN S C ARNOLD

INTENTIONALLY LEFT BLANK.